Tahmid Khan
Shengyuan Lin

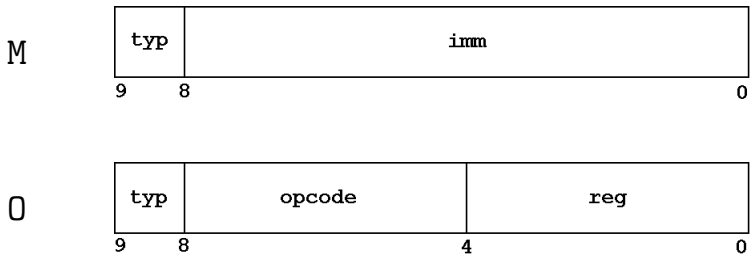
CSE 141L | EnDMe Processor Summary

1 Introduction

For this lab, our goal is to design the instruction set architecture for a 9-bit processor. Since we are only given a 9 bit processor and we need to be able to manipulate up to 256 memory locations, we decided to allocate our 9 bits as follows: 4 bits to indicate one register operand, 4 bit `opcode` to indicate which operation we will be executing this cycle, and a single bit to indicate the type of the instruction.
We decided to include this single bit instruction type indicator due to the constraint of having only 9 bits while needing to represent 256 memory locations (which requires 8 bits). Thus, we came up with 2 instruction types as indicated by the single instruction type bit. The first instruction type is the M-type which is dedicated only to reading the 256 memory locations. The second `O-type` is used for other register based operations not dealing with memory.
In conclusion, our goal for this lab was to design a preliminary ISA inspired by the Assumulator and the Mem-Reg ISAs. Our end product for this lab supports 16 registers, 16 instructions, and reading data from 256 memory locations.
We call our ISA the **En**cryption-**D**ecryption-**Me**ssage processor, or EnDMe for short

2 Instruction Formats

We introduce 2 instruction formats: the `M-type` and the `O-type`. The `M-type` is a dedicated instruction for memory access brought about due to the constraints of having to access 8-bits worth of memory spaces with only 9-bits available. The `O-type` indicates either a register operation instruction or an instruction to manipulate the accumulator register. The formats are structured as illustrated below:



3 Operations

Op	Type	Opcode	Syntax	Operation	Details
Data Transfer Operations					
save	M	----	save #imm	$acc = #imm	Loads immediate into accumulator
store	O	0000	store $reg	$reg = $acc	Loads accumulator content into $reg
lb	O	0001	lb $reg	$acc = M[$reg]	Loads a byte from the address in $reg into $acc
sb	O	0010	sb $reg	M[$reg] = $acc	Stores accumulator data into address in $reg
put	O	0011	put $reg	$acc = $reg	Loads contents of $reg into accumulator
Branching Operations					
btr	O	0100	btr	if($acc == 1) $pc = {$pc[15:8], $dst}	Jumps to label/address in $dst given $acc holds 1
jmp	O	0101	jmp	$pc = {$pc[15:8], $dst}	Unconditionally jump to the label/address in $dst
Arithmetic Operations					
add	O	0110	add $reg	$acc = $reg + $acc	Sums the contents of accumulator with $reg
sub	O	0111	sub $reg	$acc = $reg - $acc	Subtracts contents of accumulator from $reg
Bit-Oriented Operations					
and	O	1000	and $reg	$acc = $acc & $reg	Does bitwise AND and of $reg with accumulator
xor	O	1001	xor $reg	$acc = $acc $\oplus$ $reg	Does exclusive or of $reg with accumulator
sfl	O	1010	sfl $reg	$acc = $acc << $reg	Shifts contents of accumulator left by $reg bits
sfr	O	1011	sfr $reg	$acc = $acc >> $reg	Shifts contents of accumulator right by $reg bits
Logic Operations					

cmp	O	1100	cmp $reg	if($reg == $acc)
				$acc = 1
				else
				$acc = 0
					Checks the contents of $reg and the accumulator for equality and sets accumulator to 1 or 0 accordingly

gtr	O	1101	gtr $reg	if($reg > $acc)
				$acc = 1
				else
				$acc = 0
					Compares the contents of $reg against the accumulator and accordingly sets accumulator to 1 or 0 depending on which is greater

4 Internal Operands

We use 4 bits to indicate 16 registers. Of the 16 registers, we define 3 special-use registers. The last register is designated as a destination register that should not be touched by programs aside for storing a jump location prior to a `btr` instruction. This register is for the exclusive use of the Branch on True instruction `btr`; this is `$dst`. Finally, registers 13 and 14 are special preserved registers. Aside from these special registers the remaining registers are general purpose temporary registers.

Number	Name	Reg Code	Purpose
0-11	`$r1-$r12`	0000-1100	General purpose registers (Non-preserved)
13-14	`$r13-$r14`	1101-1110	Special preserved registers
15	`$dst`	1111	Dedicated destination register for BTR and JMP

5 Control Flow

We have two branching instructions, namely *branch on true* `btr` and *jump* `jmp`. Our ISA uses absolute branching to jump between different parts of the program. This forced us to optimize our code until it could fit within 255 instructions because using only 8-bits, absolute branching only covers 256 possible unique PC locations. The destination must be stored into the $DST register before either of the above instructions are called. The EnDMe ISA emulates the MIPS architecture by concatenating the 8 LSB of the location in $DST to the 8 MSB of the current PC.

6 Addressing Modes

We chose direct addressing for our purposes due to the fact that we only need to be able to access 256 memory spaces which can be represented using 8-bits. Every time we want to read or write from a memory location, we fist save its index to accumulator, followed by a load word or store word depending on whether we are accessing or storing data into memory. For example, let's say we want to get the value in memory location 128 and want to save in into memory location 129:

```
save #128      #save 128 to accumulator
store r2       #store it to r2
save #129      #save 129 to accumulator
store r3       #store it to r3
lw r2          #load M[128] into accumulator
sw r3          #store the data into M[129]
```

Tahmid Khan
Shengyuan Lin

7 Architecture Type and Example

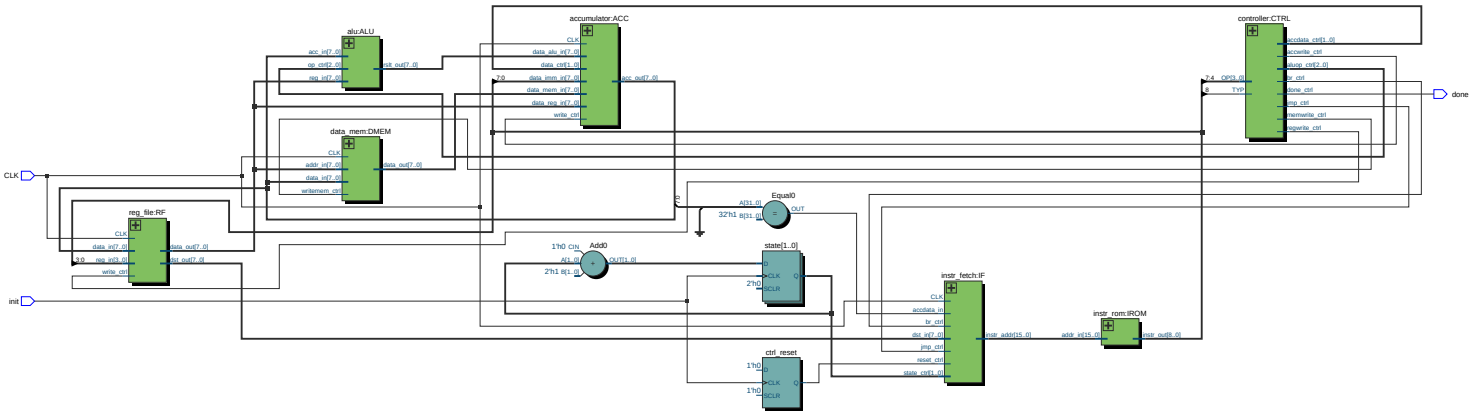
We originally intended to base our ISA on the design of an accumulator but due to the addition of several registers, our custom ISA became somewhat of a hybrid between an accumulator architecture and mem-reg architecture.

The following is a simple example of how either M- or O-type instructions should look in machine code:

Instruction	Machine Code		
`save #65`	TYPE 1	IMM 0 1 0 0 0 0 0 1	
`xor $r2`	TYPE 0	OPCODE 1 0 0 1	REG 0 0 1 0

8 RTL Block Diagram

See next page


alu:ALU
acc_in[7..0]
op_ctrl[2..0]
reg_in[7..0]
rslt_out[7..0]
accumulator:ACC
CLK
data_alu_in[7..0]
data_ctrl[1..0]
7:0
data_imm_in[7..0]
data_mem_in[7..0]
data_reg_in[7..0]
write_ctrl
acc_out[7..0]
controller:CTRL
accdata_ctrl[1..0]
accwrite_ctrl
aluop_ctrl[2..0]
br_ctrl
done_ctrl
jmp_ctrl
memwrite_ctrl
regwrite_ctrl
7:4 OP[3..0]
3:0 TYP
done
data_mem:DMEM
CLK
addr_in[7..0]
data_in[7..0]
writemem_ctrl
data_out[7..0]
CLK
reg_file:RF
CLK
data_in[7..0]
2:0 reg_in[3..0]
write_ctrl
data_out[7..0]
dst_out[7..0]
init
Equal0
A[31..0]
32'h1 B[31..0]
OUT
0
1'h0 CIN
Add0
A[1..0]
2'h1 B[1..0]
OUT[1..0]
state[1..0]
D
CLK
Q
SCLR
2'h0
ctrl_reset
1'h0
D
CLK
Q
SCLR
1'h0
instr_fetch:IF
CLK
accdata_in
br_ctrl
dist_in[7..0]
jmp_ctrl
reset_ctrl
data_ctrl[1..0]
instr_addr[15..0]
instr_rom:IROM
addr_in[15..0]
instr_out[8..0]